On the Efficacy and High-Performance Implementation of Quaternion Matrix Multiplication

DAVID B. WILLIAMS–YOUNG*, Lawrence Berkeley National Laboratory
XIAOSONG LI, University of Washington

Quaternion symmetry is ubiquitous in the physical sciences. As such, much work has been afforded over the years to the development of efficient schemes to exploit this symmetry using real and complex linear algebra. Recent years have also seen many advances in the formal theoretical development of explicitly quaternion linear algebra with promising applications in image processing and machine learning. Despite these advances, there do not currently exist optimized software implementations of quaternion linear algebra. The leverage of optimized linear algebra software is crucial in the achievement of high levels of performance on modern computing architectures, and thus provides a central tool in the development of high-performance scientific software. In this work, a case will be made for the efficacy of high-performance quaternion linear algebra software for appropriate problems. In this pursuit, an optimized software implementation of quaternion matrix multiplication will be presented and will be shown to outperform a vendor tuned implementation for the analogous complex matrix operation. The results of this work pave the path for further development of high-performance quaternion linear algebra software which will improve the performance of the next generation of applicable scientific applications.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**.

Additional Key Words and Phrases: quaternion, linear algebra, matrix multiplication, basic linear algebra subprograms

ACM Reference Format:
David B. Williams–Young and Xiaosong Li. 20xx. On the Efficacy and High-Performance Implementation of Quaternion Matrix Multiplication. *ACM Trans. Math. Softw.* 0, 0, Article 0 (March 20xx), 24 pages. https://doi.org/xx.xxxx/xxxxxx

1 INTRODUCTION

In the ever evolving ecosystem of high–performance computing (HPC), the full exploitation of contemporary computational resources must constitute a central research effort in computationally intensive fields such as scientific computing. However, it is often the case that simply applying conventional algorithms and data structures to existing problems will yield sub-optimal time–to–solution and resource management on modern HPC systems. By exploiting the symmetry of a particular problem and explicitly considering the structure and resources of these computational architectures, one can often develop more optimal computational research pathways. Such development has the potential to enable routine inquiry and simulation of systems which were inaccessible or impractical by existing computational methods.

*Corresponding Author

Authors' addresses: David B. Williams–Young, dbwy@lbl.gov, Lawrence Berkeley National Laboratory, Berkeley, California, 94720; Xiaosong Li, xsli@u.washington.edu, University of Washington, Seattle, Washington, 98195.

0098-3500/20xx/3-ART0 $15.00
https://doi.org/xx.xxxx/xxxxxx

In this work, we consider the computational benefits of exploiting the scalar and linear algebras generated by the quaternion numbers. The quaternions, also known as Hamilton's quaternions, are a hyper-complex number system which extends the complex numbers and are isomorphic with the special unitary group, SU(2) [Hamilton 1866]. Perhaps the most notable feature of the quaternion numbers is the loss of scalar commutivity under multiplication. As such, the algebra which is generated by the quaternions is peculiar in that it more closely resembles that of matrix algebra than that of its real or complex counterpart. Since their inception, the quaternions have seen extensive application both in pure [Cayley 1845; Chevalley 1996; Frobenius 1878; Hopf 1931; Hurwitz 1922] and applied mathematics [Arnol'd 1995; Deavours 1973; Grubin 1970; Mocoroa et al. 2006; Sudbery 1979]. Historically, the quaternions have been applied most successfully in the treatment of rigid body mechanics due to their relationship with SU(2) (and thus the group of spacial rotations, SO(3)) [Grubin 1970; Mocoroa et al. 2006]. This application has been widespread in the field of computer graphics to accelerate video animation via algorithms such as Slerp [Shoemake 1985]. From a computational perspective, quaternion arithmetic offers an attractive alternative to complex arithmetic in that it admits a higher arithmetic intensity and smaller memory footprint than that of the the complex matrix algebra it represents. A demonstration of this state of affairs will be given in the body of this work.

In the context of linear algebra, quaternions and quaternion linear algebra naturally manifest in many scientific applications such as quantum chemistry and nuclear physics [Armbruster 2017; Ekström et al. 2006; Henriksson et al. 2005; Konecny et al. 2018; Nakano et al. 2017; Peng et al. 2009; Saue et al. 1997; Saue and Helgaker 2002; Visscher and Saue 2000]. Traditionally, quantum mechanics and quantum field theories have been formulated in terms of the complex numbers. However, since the earliest days of the development of quantum mechanics, it has been known that the spinor nature of the fermionic wave function (i.e. electrons, neutrinos, etc) admits a quaternion representation due to its relationship with SU(2) [Adler 1995; Horwitz and Biedenharn 1984]. Typically, this representation manifests as a result of time–reversal being a global symmetry of the Hamiltonian [Dongarra et al. 1984; Saue et al. 1997; Stuber and Paldus 2003]. As such, the quaternion algebra has been exploited in the development of efficient real [Dongarra et al. 1984] and complex [Shiozaki 2017] eigensolvers for time–reversal symmetric Hamiltonians. Despite the success and power of these methods, their exploitation of the quaternion algebra is *implicit* in that the final computer implementation of these methods is done in either real or complex matrix arithmetic; thus they cannot leverage the full computational potential of the quaternion arithmetic. In this work, a case will be made for *explicit* exploitation of quaternion arithmetic in high–performance software.

In general, high–performance methods for scientific applications rely on highly tuned numerical linear algebra software libraries which implement the BLAS [Blackford et al. 2002; Dongarra et al. 1990, 1988; Lawson et al. 1979] and LAPACK [Anderson et al. 1999] standards for performance on modern HPC systems. Historically, numerical linear algebra has been the archetypal example and motivation for the careful consideration of a computer's architecture in the development of high–performance software [Agarwal et al. 1994; Goto and van de Geijn 2008; Gunnels et al. 2001; Whaley and Dongarra 1998; Whaley et al. 2001]. It was realized early on that straightforward implementations of operations such as matrix multiplication will yield sub-optimal performance results and that achieving peak performance on modern architectures requires a drastic departure from conventional implementations. We refer the reader to the work of [Goto and van de Geijn 2008] for a reasonably contemporary discussion on the optimization of BLAS functions, specifically matrix-matrix multiplication, on modern architectures. BLAS and LAPACK optimization still constitutes a major research effort in the field of numerical linear algebra, and this has led to a number of different approaches which are available in open source [Low et al. 2016; Van Zee and

Smith 2017; Van Zee et al. 2016; Van Zee and van de Geijn 2015; Wang et al. 2013; Whaley and Dongarra 1998; Xianyi et al. 2012] and vendor tuned software such as the Intel Math Kernel Library (MKL) and the IBM Engineering Scientific Subroutine Library (ESSL).

We note that over the years, there have been many important theoretical developments in the field of quaternion linear algebra [Rodman 2014; Zhang 1997]. Many of necessary algorithmic building blocks for ubiquitous operations such as eigenvalue and singular value decompositions, and matrix factorizations have been developed [Baker 1999; Bunse-Gerstner et al. 1989; Janovská and Opfer 2003; Jia et al. 2018; Li et al. 2019; Loring 2012; Zhang 1997]. Such explorations have been key in the development of recent methods for efficient signal and image processing by exploiting the quaternion algebra [Ell et al. 2014; Le Bihan and Mars 2004; Le Bihan and Sangwine 2003; Zeng et al. 2016]. Despite these successes, the field of quaternion linear algebra is still in its infancy in terms of software adoption by scientists and engineers. This is primarily due to the fact that, relative to its complex counterpart, there do not currently exist highly optimized software implementations of quaternion linear algebra. In order for quaternion linear algebra to become a viable alternative to complex linear algebra, such software must be developed. In this work, it will be demonstrated that optimized implementations of quaternion linear algebra operations hold the potential for leveraging drastic performance improvements relative to analogous complex operations in problems which they may be applied.

A key building block of optimized linear algebra algorithms is that of an optimized implementation of matrix multiplication, which we will refer to as GEMM in this work. Having an optimized GEMM implementation is a necessary (but not necessarily sufficient) condition for optimization more involved linear algebra operations such as eigenvalue decomposition and matrix factorization. As such, this work will focus on the performance and implementation of a quaternion GEMM to demonstrate the efficacy of high-performance quaternion linear algebra.

The remainder of this work will be organized as follows. Section 2 will review the necessary theory for the development of quaternion linear algebra and how one might leverage this algebra as an alternative to complex linear algebra for a special class of complex matrices. Section 3 will briefly review the nature and abstract structure of high–performance GEMM implementations. Section 4 details an implementation of a high-performance GEMM operation using explicitly quaternion arithmetic on a contemporary computing architecture. Finally, performance results for the implementation quaternion GEMM relative to a vendor tuned complex implementation will be presented in Sec. 5, and Sec. 6 will provide an examination of the future research which will be enabled as a result of our findings.

1.1 Notation

Throughout the remainder of this work, we will adopt the following notation conventions:

(1) Scalars of a ring $\mathbb{F}$ will be denoted with lower case letters $a \in \mathbb{F}$.
(2) $\mathbb{M}_{M,N}(\mathbb{F})$ will denote the ring of M-by-N matrices over $\mathbb{F}$. Further, $\mathbb{M}_N(\mathbb{F}) \equiv \mathbb{M}_{N,N}(\mathbb{F})$ for brevity. Matrices will be denoted with capital letters, $A \in \mathbb{M}_{M,N}(\mathbb{F})$.
(3) For $A \in \mathbb{M}_{M,N}(\mathbb{F})$, $\overline{A}$ will denote the conjugate of A, A^T its transpose, and $A^* = \overline{A}^T$ will denote its conjugate transpose. For $a \in \mathbb{F}$, we note that $\overline{a} \equiv a^*$.
(4) For $N = 1$, we denote the set of vectors over a ring as $\mathbb{V}_M(\mathbb{F})$, and denote its elements as lower-case letters, $\vec{a} \in \mathbb{V}_M(\mathbb{F})$.
(5) $A([\mu_1, \mu_2], :) \in \mathbb{M}_{\mu_2-\mu_1,K}(\mathbb{F})$ represents the sub-matrix consisting of the μ_1-st to μ_2-nd rows of A. If $\mu_2 - \mu_1$ is to be understood from the context, we use the abbreviated notation $A_{(\mu_1)} \equiv A([\mu_1, \mu_2], :)$.

(6) $A(:,[\nu_1,\nu_2]) \in \mathbb{M}_{M,\nu_2-\nu_1}(\mathbb{F})$ represents the sub-matrix consisting of the ν_1-st to ν_2-nd columns of A. If $\nu_2 - \nu_1$ is to be understood from the context, we use the abbreviated notation $A^{(\nu_1)} \equiv A(:,[\nu_1,\nu_2])$.
(7) The limiting cases of single row or column vectors will be denoted $\vec{a}_{(\mu)} \equiv A(\mu,:)$ and $\vec{a}^{(\nu)} \equiv A(:,\nu)$.
(8) If the indices μ_3 and μ_4 are to be understood from the context, we denote the sub-matrix of a sub-matrix as $A^{(\mu_1,\mu_3)} = A^{(\mu_1)}([\mu_3,\mu_4],:)$, and so on for row sub-matrices.

2 THE QUATERNION ALGEBRA

2.1 Scalar Operations

Fundamental to the development of quaternion linear algebra is the nature of the scalar algebra generated by the quaternions. In this work, the set of quaternion numbers will be denoted $\mathbb{H}$ and is defined as the set of all q that [Hamilton 1866]

$$q = q^0 e_0 + q^1 e_1 + q^2 e_2 + q^3 e_3, \qquad q^0, q^1, q^2, q^3 \in \mathbb{R}, \tag{1}$$

where q^0 is referred to as the *scalar* component of q and $\{q^1, q^2, q^3\}$ is referred to as its *vector* component. $\{e_0, e_1, e_2, e_3\}$ are the quaternion basis elements and they generate the skew–symmetric algebra defined by,

$$e_0 e_j = e_j e_0 = e_j, \qquad j \in \{0,1,2,3\}, \tag{2a}$$

$$e_i e_j = -\delta_{ij} e_0 + \sum_{k=1}^{3} \varepsilon_{ij}^k e_k, \qquad i,j \in \{1,2,3\}, \tag{2b}$$

where δ is the Kronecker delta and ε is the totally anti-symmetric Levi-Civita tensor. As such, the following must hold true

$$e_i e_j = -e_j e_i, \qquad i,j \in \{1,2,3\},\ i \neq j, \tag{3a}$$

$$e_1 e_2 e_3 = -e_0. \tag{3b}$$

Given Eq. (2), the product of quaternion scalars $p, q \in \mathbb{H}$ is given by the Hamilton product

$$pq = \left(p^0 q^0 - \sum_{i=1}^{3} p^i q^i\right) e_0 + \sum_{k=1}^{3} \left(p^0 q^k + p^k q^0 + \sum_{i,j=1}^{3} \varepsilon_{ij}^k p^i q^j\right) e_k, \tag{4}$$

and is thus non-commutative

$$[p,q] \equiv pq - qp = \sum_{i,j,k=1}^{3} \varepsilon_{ij}^k \left(p^i q^j - p^j q^i\right) e_k. \tag{5}$$

There are a number remarkable results that arise from the algebra defined by Eqs. (1) and (2) which are important to the construction of quaternion linear algebra. The first is that $\mathbb{H}$ constitutes a normed division algebra, and is in fact the largest normed division algebra for which multiplication is associative [Frobenius 1878]. As such, we may define a quaternion norm and inverse for every nonzero element of $\mathbb{H}$,

$$\|q\| = \sqrt{\sum_{i=0}^{3} (q^i)^2}, \tag{6a}$$

$$q^{-1} = \frac{\overline{q}}{\|q\|} \quad \forall q \neq 0, \tag{6b}$$

where we have defined the quaternion conjugate

$$\overline{q} \equiv q^* = q^0 e_0 - q^1 e_1 - q^2 e_2 - q^3 e_3. \tag{7}$$

We note here that quaternions of unit norm ($\|q\| = 1$) are known in the literature as *versors* [Hamilton 1866]. In examining Eqs. (6) and (7), the expressions for norm, inverse and conjugate closely resemble those of the complex numbers, $\mathbb{C}$. In fact, $\mathbb{C}$ is a sub-algebra embedded in $\mathbb{H}$, and this relationship is crucial for the development of the relationship between complex and quaternion linear algebra.

To examine the relationship between $\mathbb{C}$ and $\mathbb{H}$, we introduce a common, simplified notation

$$q = \underline{q}^0 + \underline{q}^1 e_2, \tag{8}$$

where

$$\underline{q}^0 = q^0 e_0 + q^1 e_1, \tag{9a}$$

$$\underline{q}^1 = q^2 e_0 + q^3 e_1. \tag{9b}$$

Consider the subset $\underline{\mathbb{C}} \subset \mathbb{H}$ defined by

$$\underline{\mathbb{C}} = \left\{ q \in \mathbb{H} \quad | \quad q^2 = q^3 = 0 \right\}. \tag{10}$$

Note that $\underline{q}^0, \underline{q}^1 \in \underline{\mathbb{C}}$. The algebra defined by $\underline{\mathbb{C}}$ is exactly that of $\mathbb{C}$ (this may be easily verified through expansion of Eq. (4)). Thus, $\underline{\mathbb{C}} \cong \mathbb{C}$ via the map

$$\underline{q} = q^0 e_0 + q^1 e_1 \quad \longleftrightarrow \quad z = q^0 + q^1 i, \tag{11}$$

with $\underline{q} \in \underline{\mathbb{C}}$ and $z \in \mathbb{C}$. It is important to note here that Eq. (11) does not imply $e_0 = 1$ nor $e_1 = i$, simply that there exists a bijection between $\mathbb{C}$ and $\underline{\mathbb{C}}$. Keeping this in mind, however, it will typically be the case that one can use them interchangeably without ambiguity. As such, whether scalars of the form given in Eq. (10) are treated as complex or quaternion scalars should will be implied from their context in the following discussion.

While one tends to describe quaternions in terms of scalars (and rightly so), the algebra which they generate more closely that of a matrix algebra, specifically that of a Lie group [Hall 2015]. In the development of quaternion linear algebra, it is instructive to examine the isomorphism between the versors and the special unitary group SU(2) through the mapping of basis elements

$$e_0 \mapsto \sigma_0, \quad e_1 \mapsto i\sigma_3, \quad e_2 \mapsto i\sigma_2, \quad e_3 \mapsto i\sigma_1, \tag{12}$$

where the Pauli matrices are given as

$$\sigma_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{13}$$

By expanding in terms of the Pauli basis, it may be demonstrated that the algebra of $\mathbb{H}$ is isomorphic to the algebra generated by $\langle \mathrm{SU}(2) \rangle \subset \mathbb{M}_2(\mathbb{C})$ via the map

$$q \in \mathbb{H} \quad \mapsto \quad q_{\mathbb{C}} = \begin{bmatrix} \underline{q}^0 & \underline{q}^1 \\ -\overline{\underline{q}}^1 & \overline{\underline{q}}^0 \end{bmatrix} \in \mathbb{M}_2(\mathbb{C}). \tag{14}$$

Here we have denoted the complex matrix representation of a quaternion scalar with a subscript $\mathbb{C}$.

While the representation given in Eq. (14) may seem inconsequential, it demonstrates the great potential for improving computational performance by exploiting quaternion arithmetic. As their

Table 1. Real floating point operations (FLOPs) comparison for elementary arithmetic operations using $\mathbb{H}$ and $\mathbb{M}_2(\mathbb{C})$ data structures. Note that FLOP counts for $\mathbb{M}_2(\mathbb{C})$ consider a generic complex matrix and assume no additional structure.

Operation	FLOPs in $\mathbb{H}$	FLOPs in $\mathbb{M}_2(\mathbb{C})$
Addition (Eq. (15a))	4	8
Multiplication (Eq. (15b))	16	32

algebras are isomorphic, it is known that

$$p + q \quad \longleftrightarrow \quad p_{\mathbb{C}} + q_{\mathbb{C}}, \tag{15a}$$

$$pq \quad \longleftrightarrow \quad p_{\mathbb{C}} q_{\mathbb{C}}, \tag{15b}$$

Although the result of both the quaternion and complex arithmetic may be thought of as to represent the same mathematical object (up to an isomorphism), the computational work required for these operations is different for the two arithmetics, respectively. Table 1 summarizes the number of *real* floating point operations (FLOPs) required for the operations given in Eq. (15). In this work we adopt the convention that the operation

$$a = a + bc, \qquad a, b, c \in \mathbb{R}, \tag{16}$$

constitutes a single FLOP. From Tab. 1, we can see that there is a 2x reduction in FLOPs for $\mathbb{H}$–arithmetic over generic $\mathbb{M}_2(\mathbb{C})$–arithmetic for the same mathematical operation. We may further note that there is also a 2x reduction in memory operations (MOPs) and computational storage requirements between $\mathbb{H}$ and $\mathbb{M}_2(\mathbb{C})$ data structures. This fact will prove important in the following developments of high–performance quaternion linear algebra.

As $\mathbb{H}$ forms a normed, associative division algebra, it is natural to consider quaternion linear algebra, i.e. the algebra generated by matrices and vectors of *quaternion* elements, as an extension of the discussion presented in this subsection. In the following subsection, we briefly review the relevant theory to motivate the usage of explicitly quaternion linear algebra for software implementation.

2.2 Matrices of Quaternions and Quaternion Linear Algebra

Consider the space of $M \times N$ matrices with quaternion elements denoted $\mathbb{M}_{M,N}(\mathbb{H})$ and given by the generic element $Q \in \mathbb{M}_{M,N}(\mathbb{H})$ [Zhang 1997],

$$Q = Q^0 e_0 + Q^1 e_1 + Q^2 e_2 + Q^3 e_3, \quad Q^0, Q^1, Q^2, Q^3 \in \mathbb{M}_{M,N}(\mathbb{R}). \tag{17}$$

Note the similarity of Eq. (17) with Eq. (1). In analogy with $\mathbb{M}_{M,N}(\mathbb{C})$, we may define conjugate and conjugate transpose operations on $\mathbb{M}_{M,N}(\mathbb{C})$ via

$$(\overline{Q})_{\mu\nu} = \overline{Q_{\mu\nu}}, \tag{18a}$$

$$(Q^*)_{\mu\nu} = \overline{Q_{\nu\mu}}. \tag{18b}$$

In the same manner as $\mathbb{M}_N(\mathbb{C})$, we define quaternion hermiticity as $Q = Q^*$. Further, we may define a scalar and matrix product operations for $P \in \mathbb{M}_{M,K}(\mathbb{H})$, $Q \in \mathbb{M}_{K,N}(\mathbb{H})$ and $q \in \mathbb{H}$ [Zhang 1997].

$$(qQ)_{\mu\nu} = qQ_{\mu\nu}, \tag{19a}$$

$$(qPQ)_{\mu\nu} = q \sum_{\kappa=1}^{K} P_{\mu\kappa} Q_{\kappa\nu}. \tag{19b}$$

However, unlike real and complex matrices, the loss of scalar commutivity in $\mathbb{H}$ dictates that we must also consider operations of the form

$$(Qq)_{\mu\nu} = Q_{\mu\nu}q, \tag{20a}$$

$$(PqQ)_{\mu\nu} = \sum_{\kappa=1}^{K} P_{\mu\kappa}qQ_{\kappa\nu}, \tag{20b}$$

where, in general, $qQ \neq Qq$ and $qPQ \neq PqQ \neq PQq$. In addition, generally $PQ \neq QP$, however this is in perfect analogy with real and complex linear algebra. As one may intuitively guess, this loss of scalar commutivity greatly complicates proofs and algorithm development in quaternion linear algebra [Rodman 2014; Zhang 1997], often requiring researchers to resort to rather complex and abstract mathematical paradigms, such as algebraic topology [Baker 1999; Zhang 1997], to obtain the desired outcomes. Despite these complications, it is possible to extend operations which are important to scientific application, such as matrix inversion [Loring 2012; Zhang 1997] and eigenvalue decomposition [Baker 1999; Bunse-Gerstner et al. 1989; Jia et al. 2018; Li et al. 2019; Zhang 1997], to $\mathbb{M}_N(\mathbb{H})$. In particular, the set of all invertable quaternion matrices forms a group under the matrix product [Zhang 1997].

Just as $\mathbb{H}$ admits a close relationship with $\mathbb{C}$ and $\mathbb{M}_2(\mathbb{C})$, analogous relationships may be developed between $\mathbb{M}_{M,N}(\mathbb{H})$, $\mathbb{M}_{M,N}(\mathbb{C})$ and $\mathbb{M}_{2M,2N}(\mathbb{C})$. Consider the subset $\mathbb{M}_{M,N}(\underline{\mathbb{C}}) \subset \mathbb{M}_{M,N}(\mathbb{H})$,

$$\mathbb{M}_{M,N}(\underline{\mathbb{C}}) = \left\{ Q \in \mathbb{M}_{M,N}(\mathbb{H}) \quad | \quad Q^2_{\mu\nu} = Q^3_{\mu\nu} = 0 \right\}. \tag{21}$$

We may define an analogous expression to Eq. (8) for $\mathbb{M}_{M,N}(\mathbb{H})$ via

$$Q = \underline{Q}^0 + \underline{Q}^1 e_2, \tag{22a}$$

$$\underline{Q}^0 = Q^0 e_0 + Q^1 e_1, \tag{22b}$$

$$\underline{Q}^1 = Q^2 e_0 + Q^3 e_1, \tag{22c}$$

with $\underline{Q}^0, \underline{Q}^1 \in \mathbb{M}_{M,N}(\underline{\mathbb{C}})$. In the same manner as $\underline{\mathbb{C}} \cong \mathbb{C}$ (Eq. (11)), $\mathbb{M}_{M,N}(\underline{\mathbb{C}}) \cong \mathbb{M}_{M,N}(\mathbb{C})$ via the map

$$\underline{Q} = Q^0 e_0 + Q^1 e_1 \quad \longleftrightarrow \quad Z = Q^0 + Q^1 i. \tag{23}$$

To construct its relationship to $\mathbb{M}_{2M,2N}(\mathbb{C})$, we examine the $\mathbb{M}_2(\mathbb{C})$ representation of a quaternion matrix element,

$$\left(Q_{\mu\nu}\right)_{\mathbb{C}} = Q^0_{\mu\nu}\sigma_0 + iQ^1_{\mu\nu}\sigma_3 + iQ^2_{\mu\nu}\sigma_2 + iQ^3_{\mu\nu}\sigma_1. \tag{24}$$

Thus Eq. (24) may be written in terms of a Kronecker product:

$$\begin{aligned} Q_{\mathbb{C}} &= Q^0 \otimes \sigma_0 + Q^1 \otimes i\sigma_3 + Q^2 \otimes i\sigma_2 + Q^3 \otimes i\sigma_1 \\ &= \begin{bmatrix} \underline{Q}^0 & \underline{Q}^1 \\ -\overline{\underline{Q}}^1 & \overline{\underline{Q}}^0 \end{bmatrix} \in \mathbb{M}_{2M,2N}(\mathbb{C}), \end{aligned} \tag{25}$$

where we have denoted the complex matrix representation of the quaternion matrix with a subscript $\mathbb{C}$ in analogy with Eq. (14).

In analogy to Eq. (15), the isomorphism between $\mathbb{M}_{M,N}(\mathbb{H})$ and $\mathbb{M}_{2M,2N}(\mathbb{C})$ admits the following relationships

$$P + Q \quad \longleftrightarrow \quad P_{\mathbb{C}} + Q_{\mathbb{C}}, \tag{26a}$$

$$PQ \quad \longleftrightarrow \quad P_{\mathbb{C}}Q_{\mathbb{C}}. \tag{26b}$$

As in Eq. (15), the amount of computational work required to perform the operations in Eq. (26) in quaternion and complex arithmetic are different. As an extension of Tab. 1, Tab. 2 summarizes

Table 2. Real floating point operations (FLOPs) comparison for common linear algebra operations using $\mathbb{M}_N(\mathbb{H})$ and $\mathbb{M}_{2N}(\mathbb{C})$ data structures. As in Tab. 1, FLOP counts for $\mathbb{M}_{2N}(\mathbb{C})$ consider a generic complex matrix and assume no additional structure.

Operation	FLOPs in $\mathbb{M}_N(\mathbb{H})$	FLOPs in $\mathbb{M}_{2N}(\mathbb{C})$
Addition (Eq. (26a))	$4N^2$	$8N^2$
Multiplication (Eq. (26b))	$16N^3$	$32N^3$

differences in the the number of FLOPs required for the same algebraic operation in the two arithmetics, respectively. For simplicity and brevity, the summary in Tab. 2 only accounts for $\mathbb{M}_N(\mathbb{H})$ and $\mathbb{M}_{2N}(\mathbb{C})$, though completely analogous results hold for the general rectangular case. Just as in Tab. 1, there is a 2x reduction in both FLOPs and MOPs in utilizing explicitly quaternion arithmetic and data structures over the analogous complex operations. However, this comparison is in terms of a *ratio* of computational work requirements. In terms of raw differences between the two arithmetics, the potential computational savings scale to some power of the dimension the matrix in question. For example, the difference in the number of FLOPs required for quaternion / complex matrix multiplication is $16N^3$. For small N this would not make a drastic difference, but for large N, this difference becomes significant. Due to the fundamental and central importance of matrix multiplication in numerical linear algebra, similar comparisons could be made for any matrix operation, such as eigenvalue decomposition or matrix factorization, between complex and quaternion arithmetic.

3 HIGH–PERFORMANCE MATRIX–MULTIPLICATION

The cornerstone of high–performance numerical linear algebra software is the optimized implementation of general matrix–matrix multiplication (GEMM). Without an optimized GEMM implementation, operations such as eigenvalue decomposition and matrix inversion become impractical for large matrices. Thus, the first step in the development of high-performance quaternion linear algebra is the development of an optimized quaternion GEMM.

Over the past several decades, an enormous amount of research effort in the fields of numerical linear algebra and HPC has been directed towards the development of optimized GEMM operations on various computing architectures. As a result, many different strategies have been developed for high–performance GEMM implementations [Goto and van de Geijn 2008; Gunnels et al. 2001; Van Zee and van de Geijn 2015; Wang et al. 2013; Whaley and Dongarra 1998; Xianyi et al. 2012]. Despite their differences, the common motif among these methods is the rejection of a "one–size–fits–all" development strategy for all computing platforms, i.e. one must explicitly consider and optimize for the underlying features of the computer architecture in question to reach optimal performance. In modern HPC, there are effectively three fundamental aspects of computing architectures which must be considered in the development of optimized GEMM operations [Goto and van de Geijn 2008]:

(1) Efficient and effective utilization of various levels of the computational data and instruction caches.
(2) Utilization of microarchitechture specific features such as single instruction multiple data (SIMD) and fused multiply–add (FMA) operations.
(3) Achieving efficient parallelism on modern multi–core and many–core computing architectures.

To demonstrate the efficacy of quaternion GEMM, we will only consider the former two of these features; leaving the treatment of parallelism for future work. Due to its relative simplicity and

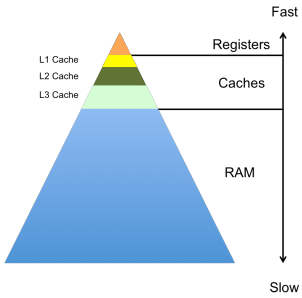


Fig. 1. A simplified model of the memory hierarchy on modern computing architectures.

portability to general architectures, the development of high–performance quaternion GEMM operations in this work will extend the strategy adopted by the BLIS library for real and complex GEMM operations [Low et al. 2016; Van Zee and Smith 2017; Van Zee et al. 2016; Van Zee and van de Geijn 2015]. In the BLIS strategy, the aspects of the GEMM operation which must be explicitly optimized for a specific architecture are factored into a manageably small set of auxiliary procedures, referred to as *kernels*, while the general scaffold for the GEMM remains consistent between architectures. Further, the structure and function of the kernels yielded by this strategy are designed in such a way that they may be used in the implementation of other BLAS-3 functionality such as rank-k updates (XSYRK), triangular matrix multiplication (XTRMM), etc. In this section, we examine the salient aspects of this strategy which are agnostic to the data representation and arithmetic operations relating to the matrices being multiplied.

3.1 The General Algorithm

Consider the GEMM of two matrices $A \in \mathbb{M}_{M,K}(\mathbb{F})$, $B \in \mathbb{M}_{K,N}(\mathbb{F})$ over a general ring $\mathbb{F}$,

$$C = \alpha AB + \beta C, \tag{27}$$

where $\alpha, \beta \in \mathbb{F}$ and $C \in \mathbb{M}_{M,N}(\mathbb{F})$. Computationally, A, B and C are stored as linear, contiguous data structures of lengths MK, KN and MN, respectively. In this work, we will consider column–major storage of matrices, i.e. $A_{\mu\kappa}$ and $A_{(\mu+1)\kappa}$, and $A_{M\kappa}$ and $A_{1(\kappa+1)}$ are stored contiguously in memory.

For comparison in the following, Algorithm 1 outlines the simplest implementation of the GEMM operation which was suggested in the earliest developments of the BLAS standard [Dongarra et al. 1990]. This method will be referred to as the *reference* GEMM algorithm. To fully understand the drawbacks of Algorithm 1 and to motivate the development of a more optimal algorithm, we must examine that nature of the memory hierarchy on modern computers. Figure 1 illustrates a simplified model of a representative memory hierarchy on a modern computer [Goto and van de Geijn 2008]. At the top of the hierarchy, the fastest and least abundant memory resource are the registers which physically reside on the processor. It is on the data which resides in the registers that the processor may issue instructions such as arithmetic operations, etc. On architectures which support SIMD instructions, i.e vector processors with instruction sets such as SSE, AVX, AVX2 and AVX-512, each floating point register can hold a small number of floating point numbers at a time, typically between 2 and 16. However, their abundance is very limited: 16 registers on SSE, AVX and AVX2, and 32 on AVX-512. Thus, to fully exploit the speed of the registers, care must be taken to carefully populate the data which resides there to minimize the data movement between the registers and other levels of the hierarchy.

At the bottom of the hierarchy is the slowest and largest memory resource: the random access memory (RAM). It is in the RAM that the matrices which participate in the GEMM operation are typically stored. The RAM is the memory resource that resides furthest from the processor, which allows it to be orders of magnitude larger than any of the other memory resources (on the order of 1GB-1TB). However, the penalty for its size and distance from the processor is a high access latency. The speed at which data can be moved to and from RAM varies drastically between different architectures and manufacturers, and also depends on factors such as how the data being moved is laid out in memory (contiguous, strided, cache aligned, etc). Generally, reading to and from RAM amounts to hundreds of clock cycles on modern processors. Due to its very high latency, data movement in and out of RAM must be kept to a minimum to achieve optimal performance. As such data is rarely read directly from RAM to the registers or visa versa. Instead, it is typically the case that data is read to and from the RAM to a low latency intermediary storage, known as the data cache, which resides closer to the processor and is thus capable of moving data to and from the registers much faster than would be possible from the RAM.

Due to the slow rate at which RAM may be accessed, typical memory access patterns dictate that the RAM should be read in large chunks of contiguous data into the cache. Whenever a read instruction is issued from the processor for a particular memory address in RAM, it first checks if that data resides in cache. If the data resides in cache, what is referred to as a *cache hit*, it may be read directly from cache and avoid the RAM completely. However, if the data does not reside in cache when the instruction is issued, the data must be moved from RAM to cache and then read into the registers. This process is referred to as a *cache miss*. Due to the large latency differential between RAM and cache, the penalty for a cache miss can often be quite large. Further, the restricted size of the cache only allows a limited amount of data can be stored there at any point in time. When the cache reaches its capacity, data which resides in the cache must be replaced when new data is read in from the RAM. The process by which this replacement happens is referred to as the cache's replacement policy. If data is to be often reused in an algorithm, it is important to ensure that it resides in the cache as often as possible to minimize the probability of a cache miss. Thus, knowledge of the replacement policy is paramount in the development of a strategy for cache

Algorithm 1: Reference GEMM Algorithm

Input : Matrices $A \in \mathbb{M}_{M,K}(\mathbb{F}), B \in \mathbb{M}_{K,N}(\mathbb{F}), C \in \mathbb{M}_{M,N}(\mathbb{F})$,
Scalars $\alpha, \beta \in \mathbb{F}$

Output: $C = \alpha AB + \beta C$

for $\nu = 1 : N$ **do**
1 Load $\vec{c}^{(\nu)} = C(:, \nu)$ into cache
2 $\vec{c}^{(\nu)} = \beta \vec{c}^{(\nu)}$
for $\kappa = 1 : K$ **do**
3 Load $\vec{a}^{(\kappa)} = A(:, \kappa)$ into cache
4 Load $B_{\kappa\nu}$
5 $\vec{c}^{(\nu)} = \vec{c}^{(\nu)} + \alpha \vec{a}^{(\kappa)} B_{\kappa\nu}$
end
6 Store $\vec{c}^{(\nu)}$
end

population to maximally reuse the data the resides there while not ejecting reusable data with data which is to be used less often.

On contemporary architectures, the cache is divided into cascading "levels": the L1, L2, L3 caches, etc. The capacity and access latencies for the cache levels vary considerably between processor generations and manufacturers; however, the general trend is to lose an order of magnitude on access latency and gain an order of magnitude in capacity between successive cache levels. For example, the Intel(R) Xeon(R) CPU E5-2660 (Sandy Bridge) processor yields cache capacities of 32 kB, 256 kB, 20 MB and access latencies of 4, 12 and 29 clock cycles for the L1, L2 and L3 caches, respectively [Fog 2012]. It is typically the case that the population of the different levels of cache cannot be explicitly programmed; one typically relies on heuristics issued by the CPU, such as data prefetching and cache replacement policies, to perform this population. However, with knowledge of the sizes of the cache levels and replacement policies, one may develop algorithms which aim to populate these caches optimally for data reuse.

From the perspective of effective utilization of the memory hierarchy and the other aforementioned features of computing architectures, there are a number of drawbacks in Algorithm 1:

- All of A is loaded into cache for each column of C,
- For large M, K, loading A potentially ejects $\vec{c}^{(\nu)}$ from cache, triggering a cache miss on each update of $\vec{c}^{(\nu)}$,
- There is no useful caching of B,
- In a high-level programming language, this algorithm relies on an optimizing compiler to utilize SIMD, FMA, etc.
- Scalable parallelism is non–trivial.

Algorithm 1 is referred to as a *memory bound* algorithm, i.e. its performance is completely determined by the latency at which data may be moved to and from the RAM. As such, even for relatively small GEMM operations, performance will be sub-optimal [Goto and van de Geijn 2008]. A demonstration of this state of affairs in the context of quaternion GEMM will be given in Sec. 5.

In order to overcome the memory bottle neck, one must develop an algorithm which populates the levels of cache and registers with sub-matrices of A, B and C according how their data may be reused throughout the GEMM operation. For a detailed explanation of the extent to which one may reuse different sub-matrices of A, B and C, we refer the reader to the work of [Goto and van de Geijn 2008]. In general, the mechanism by which one achieves optimal cache utilization is through a layered approach to the GEMM operation [Goto and van de Geijn 2008; Gunnels et al. 2001; Van Zee and van de Geijn 2015; Whaley and Dongarra 1998]. An optimized layered GEMM algorithm may be constructed through the specification of three caching parameters: $M_c, N_c, K_c \in \mathbb{Z}^+$, two register blocking parameters: $N_r, M_r \in \mathbb{Z}^+$, two packing kernels: `PACK1`, `PACK2`, and a microkernel, `KERN`. A representative example of such an algorithm, specifically the algorithm which has been proposed in the development of the BLIS framework [Van Zee and Smith 2017; Van Zee et al. 2016; Van Zee and van de Geijn 2015], is outlined in Algorithm 2. For simplicity in Algorithm 2, we have assumed $(N \mod N_c) = (M \mod M_c) = (K \mod K_c) = 0$ and $(N_c \mod N_r) = (M_c \mod M_r) = 0$. However, extension of Algorithm 2 without these constraints is straightforward through zero padding in the packing kernels [Van Zee and van de Geijn 2015]. We note for clarity that the scaling by α in Line 10 of Algorithm 2 may instead be performed in Line 7 for rings $\mathbb{F}$ which admit scalar commutivity in the sense of Eq. (20b) (i.e. $\mathbb{R}$ and $\mathbb{C}$). Each of these parameters and kernels must be carefully chosen and optimized for each computer architecture of interest. In the following subsection, we examine the nature of each of these moieties and the factors one must consider in their selection.

Algorithm 2: General Layered GEMM Algorithm

Input : Matrices $A \in \mathbb{M}_{M,K}(\mathbb{F}), B \in \mathbb{M}_{K,N}(\mathbb{F}), C \in \mathbb{M}_{M,N}(\mathbb{F})$,
Scalars $\alpha, \beta \in \mathbb{F}$,
Caching parameters $N_c, M_c, K_c \in \mathbb{Z}^+$,
Register block sizes $N_r, M_r \in \mathbb{Z}^+$

Output : $C = \alpha AB + \beta C$

Allocate $\tilde{A}_p \in \mathbb{M}_{K_c M_r, M_c/M_r}(\mathbb{F})$, $\tilde{B}_p \in \mathbb{M}_{K_c N_r, N_c/N_r}(\mathbb{F})$
$C = \beta C$
for $\nu = 1 : N : N_c$ **do**
  Identify $C^{(\nu)} = C(:, [\nu, \nu + N_c])$
  Identify $B^{(\nu)} = B(:, [\nu, \nu + N_c])$
  for $\kappa = 1 : K : K_c$ **do**
    Identify $A^{(\kappa)} = A(:, [\kappa : \kappa + K_c])$
    Identify $B^{(\nu)}_{(\kappa)} = B^{(\nu)}([\kappa, \kappa + K_c], :)$
    Pack $\tilde{B}_p \leftarrow \mathsf{PACK2}(B^{(\nu)}_{(\kappa)})$ (L3 cache)
    for $\mu = 1 : M : M_c$ **do**
      Identify $C^{(\nu)}_{(\mu)} = C^{(\nu)}([\mu, \mu + M_c], :)$
      Identify $A^{(\kappa)}_{(\mu)} = A^{(\kappa)}([\mu, \mu + M_c], :)$
      Pack $\tilde{A}_p \leftarrow \alpha * \mathsf{PACK1}(A^{(\kappa)}_{(\mu)})$ (L2 cache)
      $j_r \leftarrow 0$
      for $\nu_r = 1 : N_c : N_r$ **do**
        Identify $C^{(\nu,\nu_r)}_{(\mu)} = C^{(\nu)}_{(\mu)}(:, [\nu_r, \nu_r + N_r])$
        Identify $\vec{b_p}^{(j_r)} = \tilde{B}_p(:, j_r)$
        $i_r \leftarrow 0$
        for $\mu_r = 1 : M_c : M_r$ **do**
          Identify $C^{(\nu,\nu_r)}_{(\mu,\mu_r)} = C^{(\nu,\nu_r)}_{(\mu)}([\mu_r, \mu_r + M_r], :)$
          Identify $\vec{a_p}^{(i_r)} = \tilde{A}_p(:, i_r)$
          $C^{(\nu,\nu_r)}_{(\mu,\mu_r)} \leftarrow \mathsf{KERN}(C^{(\nu,\nu_r)}_{(\mu,\mu_r)}, \vec{a_p}^{(i_r)}, \vec{b_p}^{(j_r)})$
          $i_r \leftarrow i_r + 1$
        end
        $j_r \leftarrow j_r + 1$
      end
    end
  end
end
Free $\tilde{A}_p, \tilde{B}_p$

Algorithm 3: Abstract Template for the Microkernel

Input : Columns $\vec{a_p} \in \mathbb{V}_{K_c M_r}(\mathbb{F})$, $\vec{b_p} \in \mathbb{V}_{K_c N_r}(\mathbb{F})$ of packed representations
$\tilde{A}_p \in \mathbb{M}_{K_c M_r, M_c/M_r}(\mathbb{F})$, $\tilde{B}_p \in \mathbb{M}_{K_c N_r, N_c/N_r}(\mathbb{F})$ of sub-matrices
$A_r \in \mathbb{M}_{M_r, K_c}(\mathbb{F})$, $B_r \in \mathbb{M}_{K_c, N_r}(\mathbb{F})$, respectively.
Sub-matrix $C_r \in \mathbb{M}_{M_r, N_r}(\mathbb{F})$.

Output : Partially updated C_r

1 Load C_r into registers.
for $\kappa = 1 : K_c$ **do**
2 Load $\vec{a_r}^{(\kappa)}$ and $\vec{b}_{r(\kappa)}$ from $\vec{a_p}$ and $\vec{b_p}$ into registers.
3 $C_r \leftarrow C_r + \vec{a_r}^{(\kappa)} \vec{b}_{r(\kappa)}$
end
4 Store C_r.

3.2 Register Blocking and The Microkernel

Consider the expression of a specific sub-matrix $C_r = C([i_1, i_2], [j_1, j_2])$ in terms of the corresponding sub-matrices $A_r = A([i_1, i_2], :)$ and $B_r = B(:, [j_1, j_2])$,

$$C_r = \sum_{\kappa=1}^{K} \vec{a_r}^{(\kappa)} \vec{b}_{r(\kappa)}. \tag{28}$$

In other words, C_r may be expressed as a sum of rank−1 updates over rows and columns of B_r and A_r, respectively. As this is the fundamental arithmetic operation of the GEMM operation to be performed by the CPU, $\vec{a_r}^{(\kappa)}$, $\vec{b}_{r(\kappa)}$ and C_r must all reside in the registers for the operation to take place. C_r is referred to as the *register block* of C, with dimensions $N_r = i_2 - i_1$ and $M_r = j_2 - j_1$. To achieve optimal memory performance, N_r and M_r must be chosen such that $\vec{a_r}^{(\kappa)}$, $\vec{b}_{r(\kappa)}$ and C_r may reside in the registers simultaneously in order to avoid data movement between the registers and other levels of the memory hierarchy [Goto and van de Geijn 2008].

In Algorithm 2, the full product, C, is constructed by successively updating each of its (disjoint) $\mathbb{M}_{M_r, N_r}(\mathbb{F})$ sub-matrices via partial summation (over K_c elements) of Eq. (28) with the microkernel performing arithmetic operations which amount to the sum over rank-1 updates. As the arithmetic kernel of the GEMM operation, the microkernel is the fundamental operation which is most sensitive to the underlying computer architecture and is a key factor in the performance of the GEMM implementation. It is in the microkernel that one must explicitly consider microarchitechture specific operations such as SIMD and FMA. As such, optimized GEMM implementations typically do not express the microkernel in a high-level language; it is typically expressed directly in assembly language [Goto and van de Geijn 2008; Van Zee and van de Geijn 2015; Whaley and Dongarra 1998] or with use of low−level access paradigms such as vector intrinsics in C++. An abstract template for a generic microkernel implementation is given in Algorithm 3.

There is a subtle, yet crucial aspect of the loop expressed in Algorithm 3 in relationship to Algorithm 2: as all of the arithmetic intensity is folded into the rank-1 updates performed from within the microkernel inner-loop, optimality of the GEMM operation is directly related to the amount of time spent in this loop. In other words, the number of operations performed inside of this loop, whether they be FLOPs or MOPs, must be kept to a minimum to achieve optimal performance. The number of FLOPs required to perform the rank-1 update is fixed based on M_r, N_r and $\mathbb{F}$, thus

optimality is generally achieved through minimizing the number of MOPs performed inside this inner loop. To this end, the microkernel utilizes packed representations, $\tilde{A}_p$ and $\tilde{B}_p$, of sub-matrices, A_r and B_r, produced by the packing kernels, PACK1 and PACK2, respectively. The remainder of this section is dedicated to the design and optimization of the packing kernels and caching parameters to achieve optimal data movement between levels of the memory hierarchy and to minimize the number of MOPs required to be performed from within the microkernel.

3.3 Sub-matrix Packing for Optimal Data Layout and Cache Utilization

Perhaps the most ingenious aspect of the layered GEMM algorithm outlined in Algorithm 2 is the utilization of auxiliary memory and packing kernels to amortize the cost of data manipulation over the movement of data between the levels of the memory hierarchy [Goto and van de Geijn 2008]. This packing strategy has two primary objectives:

(1) To populate the various levels of the cache with sub-matrices of A and B according to the extent which they will be reused in the GEMM operation as to minimize probability of triggering cache misses,
(2) To ensure optimal, contiguous data layouts of the packed sub-matrices to minimize the number of operations (FLOPs and MOPs) which must be performed from within the inner loop of the microkernel.

In the following, we will examine both of these objectives in turn.

To optimize data movement for cache utilization, one must obtain optimal choices for the caching parameters M_c, N_c and K_c for the architecture of interest. Typically, these parameters are chosen such that [Goto and van de Geijn 2008; Van Zee and van de Geijn 2015]:

- Contiguous storage of size $N_c K_c$ may reside in and be addressed from the L3 cache once the data is loaded from RAM (e.g. $\tilde{B}_p \leftarrow B^{(\nu)}_{(\kappa)}$) until it is no longer needed.
- Contiguous storage of size $M_c K_c$ may reside in and be addressed from the L2 cache once the data is loaded from RAM (e.g. $\tilde{A}_p \leftarrow A^{(\kappa)}_{(\mu)}$) until it is no longer needed.
- Contiguous storage of size $K_c N_r$ may be moved from the L3 to the L1 cache without triggering a cache miss or cache invalidation (e.g. $\vec{b_p}^{(j_r)} \leftarrow \tilde{B}_p$).

Clearly, the choice of these parameters are integrally tied to the sizes of the L1, L2 and L3 caches and the size of the data structure which represents $\mathbb{F}$. Several methods exist for determining optimal choices for the caching parameters. There has been work in the development of analytical models and formulas which take into account the specifics of $\mathbb{F}$ and the architecture in question and return optimal values for the caching parameters [Low et al. 2016]. Other approaches utilize guided or black–box optimization [Wang et al. 2013; Whaley et al. 2001; Xianyi et al. 2012], to obtain these parameters. Once these parameters have been determined, the task then becomes to develop efficient packing utilities which optimize the data layout for use with the microkernel.

There are a number of desirable features one wishes to express in the data layout of packed matrices, $\tilde{A}_p$ and $\tilde{B}_p$, to optimize the data movement between the levels of cache and the registers from within the microkernel:

- The elements of $\vec{a_r}^{(\kappa)}$ and $\vec{b}_{r(\kappa)}$ should be contiguous, respectively. As vectors, this amounts to ensuring $\vec{a}^{(\kappa)}_{r,\mu}$ and $\vec{a}^{(\kappa)}_{r,\mu+1}$ are contiguous in memory, and similarly for $\vec{b}_{r(\kappa)}$.
- The elements of $\tilde{A}_p$ and $\tilde{B}_p$ which contribute to adjacent register blocks of C should be contiguous in memory, i.e. $\vec{a_p}^{(i_r)}$ and $\vec{a_p}^{(i_r+1)}$ should be contiguous in memory.
- For $\mathbb{F}$ which is represented by a compound datatype of primitive data, e.g. $\mathbb{C}$ and $\mathbb{H}$, the primitive data for contiguous datastrutures which contain elements of type $\mathbb{F}$ should be

Algorithm 4: Abstract Template for the PACK1 Kernel

Input : Identified sub-matrix $A_r \in \mathbb{M}_{M_c,K_c}(\mathbb{F})$ (non-contiguous)

Output: Packed sub-matrix $\tilde{A}_p \in \mathbb{M}_{K_c M_r, M_c/M_r}(\mathbb{F})$ (contiguous)

for $\mu = 1 : M_c : M_r$ **do**
  for $\kappa = 1 : K_c$ **do**
1 $i \leftarrow M_r(\kappa - 1) + 1$
2 $\tilde{A}_p([i, i + M_r], \mu/M_r) \leftarrow \texttt{PACKOP1}(A_r([\mu, \mu + M_r], \kappa))$
  end
end

arranged into a data layout which allows for a minimum number of MOPs to be performed from within the microkernel, as long as map between the standard and new data layout is space preserving.

To demonstrate what is meant by a space preserving map in this context, consider an complex element, $z = a + bi \in \mathbb{C}$, which is represented by two primitive real numbers $a, b \in \mathbb{R}$ which are contiguous in memory, denoted $[a; b]$. For a datastructure which contains two contiguous elements $z_1, z_2 \in \mathbb{C}$, the data layouts $[a_1; b_1; a_2; b_2]$ and $[a_1; a_2; b_1; b_2]$ occupy the same space in memory. Thus a map between these two data layouts would be considered space preserving. While the first two aspects of data packing are well explored in the literature, the latter has not to the best of authors' knowledge. As will be demonstrated in Sec. 4, optimizing the primitive data layout of contiguous quaternion datastructures will prove important in the development of an optimized quaternion GEMM.

The fact that the rank-1 updates required by Eq. (28) and Algorithm 3 involve both row and column vectors, a single packing strategy would not be sufficient to achieve optimal data layout for both $\tilde{A}_p$ and $\tilde{B}_p$. Thus, the packing kernels PACK1 and PACK2 must be designed separately to optimize the layouts of $\tilde{A}_p$ and $\tilde{B}_p$, respectively. An abstract templates for these packing kernels are given in Algorithms 4 and 5, respectively. We refer the reader to the work of Van Zee, *et al* [Van Zee and van de Geijn 2015] for an intuitive graphical illustration of the optimal packing procedure. To account for the rearrangement of primitive data in the packing procedure, we have introduced two additional operations, PACKOP1 and PACKOP2, to perform this operation for the kernels PACK1

Algorithm 5: Abstract Template for the PACK2 Kernel

Input : Identified sub-matrix $B_r \in \mathbb{M}_{K_c,N_c}(\mathbb{F})$ (non-contiguous)

Output: Packed sub-matrix $\tilde{B}_p \in \mathbb{M}_{K_c N_r, N_c/N_r}(\mathbb{F})$ (contiguous)

for $\nu = 1 : N_c : N_r$ **do**
  for $\kappa = 1 : K_c$ **do**
1 $i \leftarrow N_r(\kappa - 1) + 1$
2 $\tilde{B}_p([i, i + N_r], \nu/N_r) \leftarrow \texttt{PACKOP2}(B_r(\kappa, [\nu, \nu + N_r]))$
  end
end

and PACK2, respectively. Note that typical implementations for real and complex GEMM would yield both PACKOP1 and PACKOP2 as either the identity or linear scaling operation.

Due to the large access latency difference between RAM and the other levels of the memory hierarchy, operations performed within PACKOP1 and PACKOP2 have little to no impact on the performance of the GEMM implementation. This is due to the fact that that these operations are to be done in the registers, and are thus amortized over the time it takes to access the data from the RAM. For example, the construction of the packed sub-matrix $\tilde{A}_p$ in Line 10 of Algorithm 2 requires the scaling of the sub-matrix $A^{(\kappa)}_{(\mu)} \rightarrow \alpha A^{(\kappa)}_{(\mu)}$. As there is a two orders of magnitude latency ratio between RAM access (O(100s) of clock cycles) and the FLOP required to scale an element of the matrix (O(4-5) clock cycles), the cost of the scaling operation may be thought of as negligible. The same logic holds true for data rearrangement operations, such as register transpose, which will be explored in the following section.

4 QUATERNION MATRIX MULTIPLICATION: HGEMM

In this section, we develop the details of a high–performance implementation of quaternion GEMM for the AVX microarchitechture. The primary focus of this section is the development of AVX-optimized versions of the kernels described in the previous section for use with quaternion arithmetic and data structures. In practice, there are two primary features of the AVX microarchitechture that one must consider in the development of optimized GEMM kernels:

(1) processors with support for AVX instructions have (at least) 16 256-bit floating point (YMM) registers, and
(2) AVX dictates support for SIMD (but not FMA) arithmetic instructions on these YMM registers.

For the purposes of this work, we will restrict the discussion of kernel development to double precision floating point storage, i.e. each floating point primitive will occupy 64-bits. As such, each YMM register on AVX can hold and perform arithmetic operations on up to 4 double precision floats, simultaneously. In analogy to the DGEMM and ZGEMM naming conventions of real and complex GEMM operations, we will refer to the double precision quaternion GEMM as HGEMM. As an extension of the standard construction of complex datatypes as two contiguous floats, the following developments will describe quaternion datatypes as four contiguous floats, $[q^0; q^1; q^2; q^3]$ using the notation of Eq. (1). As such, each AVX YMM register can hold one double precision quaternion (or equivalent) at any point in time.

4.1 Batch SIMD Quaternion Multiplication

Critical to the development of an AVX-optimized quaternion microkernel is an efficient strategy for quaternion product using SIMD arithmetic operations. The the product of quaternions given by the Hamilton product in Eq. (4) requires a minimum of 16 FLOPs to complete. As each YMM register in AVX is capable of storing and manipulating 4 floats at once, one could in principle perform some of these FLOPs concurrently if the task is simply to perform a single quaternion product. However, if the task is to perform many quaternion products in a structured manner, as is the case for the rank-1 updates required by Eq. (28), implementations which optimize for a single quaternion product will yield sub-optimal throughput. To leverage the full power of SIMD instructions in this case, one needs to develop a strategy which aims to perform multiple quaternion products simultaneously at the highest throughput possible. As each YMM register is able to manipulate 4 floats, the simplest manner to reach optimal throughput is to perform 4 quaternion products simultaneously.

Consider the batch quaternion product which takes two sets of four quaternions, $\{p_i\}_{i=1}^4$ and $\{q_i\}_{i=1}^4$, and returns a set of four quaternion products, $\{(pq)_i\}_{i=1}^4$. For simplicity in the following, we

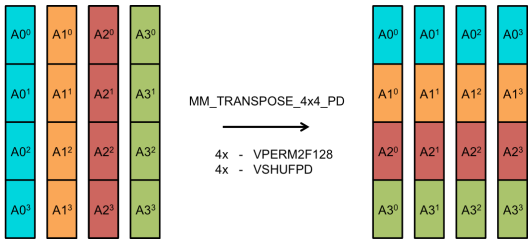


Fig. 2. Graphical illustration of a 4x4 register transpose of 4 YMM registers containing general contiguous data structures. In the general case, this operation may be completed using 4x VPERM2F128 and 4x VSHUFPD vector instructions and 4 additional YMM registers which may be used as scratch space.

will augment the product operation to perform an update of the result as opposed to an assignment,

$$\begin{bmatrix}(pq)_1\\(pq)_2\\(pq)_3\\(pq)_4\end{bmatrix} = \begin{bmatrix}(pq)_1\\(pq)_2\\(pq)_3\\(pq)_4\end{bmatrix} + \begin{bmatrix}p_1\\p_2\\p_3\\p_4\end{bmatrix} \circ \begin{bmatrix}q_1\\q_2\\q_3\\q_4\end{bmatrix} \tag{29}$$

where $\circ$ is the Hadamard (entry-wise) product such that $(pq)_1 = (pq)_1 + p_1 q_1$ and so on. Each quaternion product (Eq. (4)) may be separated into 4 sets of 4 FLOPs which update each component of the result, respectively. In the following, we examine the update of the scalar component of the product, $(pq)_i^0$, as a representative example. Note that extension and generalization to vector components of $(pq)_i$ is straightforward. The scalar components of each updated quaternion product may be obtained simultaneously by

$$\begin{bmatrix}(pq)_1^0\\(pq)_2^0\\(pq)_3^0\\(pq)_4^0\end{bmatrix} = \begin{bmatrix}(pq)_1^0\\(pq)_2^0\\(pq)_3^0\\(pq)_4^0\end{bmatrix} + \begin{bmatrix}p_1^0\\p_2^0\\p_3^0\\p_4^0\end{bmatrix} \circ \begin{bmatrix}q_1^0\\q_2^0\\q_3^0\\q_4^0\end{bmatrix} - \begin{bmatrix}p_1^1\\p_2^1\\p_3^1\\p_4^1\end{bmatrix} \circ \begin{bmatrix}q_1^1\\q_2^1\\q_3^1\\q_4^1\end{bmatrix} - \begin{bmatrix}p_1^2\\p_2^2\\p_3^2\\p_4^2\end{bmatrix} \circ \begin{bmatrix}q_1^2\\q_2^2\\q_3^2\\q_4^2\end{bmatrix} - \begin{bmatrix}p_1^3\\p_2^3\\p_3^3\\p_4^3\end{bmatrix} \circ \begin{bmatrix}q_1^3\\q_2^3\\q_3^3\\q_4^3\end{bmatrix}. \tag{30}$$

In the SIMD paradigm, each of these vectors may be represented by a single YMM register. As such, each of these Hadamard products may be performed by the VMULPD vector instruction and each vector addition (subtraction) by the VADDPD (VSUBPD) vector instruction. In this form, the entire batch quaternion multiplication may be completed using 32 vector instructions.

The structure of Eq. (30) requires that each of the sets $\{p_i\}$, $\{q_i\}$ and $\{(pq)_i\}$ occupy 4 YMM registers, with each register containing a particular quaternion component of each element in the set, respectively. In other words, one YMM register contains all of the scalar components for each element of $\{p_i\}$, one for the scalar components of $\{q_i\}$, and so on for the vector components of these sets and for the components of $\{(pq)_i\}$. For clarity in the following, we will denote the YMM register containing the scalar components of $\{p_i\}$, $\{q_i\}$ and $\{(pq)_i\}$ as P^0, Q^0 and PQ^0, respectively, and so on for the vector parts of these sets with indices 1, 2 and 3. Using this notation, we will define the SIMD implementation of Eq. (29) as

$$(\mathsf{PQ}^0, \mathsf{PQ}^1, \mathsf{PQ}^2, \mathsf{PQ}^3) \leftarrow \mathtt{HMUL}(\{\mathsf{PQ}^i\}, \{\mathsf{P}^i\}, \{\mathsf{Q}^i\}). \tag{31}$$

For quaternion data structures which store a single quaternion contiguously, such as the one considered in this work, the vector load instruction (VMOVAPD) would populate each register with the 4 components of a single quaternion. As such, one would need to rearrange the quaternion data once it is read into registers in order to utilize Eq. (31). In general, this rearrangement may be achieved by a 4x4 register transpose on each of the quaternion sets. This register transpose will be

denoted MM_4x4_TRANSPOSE_PD in the following and is illustrated graphically in Fig. 2. For clarity, we endow MM_4x4_TRANSPOSE_PD with the function signature

$$(\mathsf{P}^0, \mathsf{P}^1, \mathsf{P}^2, \mathsf{P}^3) \leftarrow \texttt{MM\_4x4\_TRANSPOSE\_PD}(\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3, \mathsf{P}_4), \tag{32}$$

where P_1 is a YMM register containing the components of p_1, P_2 the components of p_2, and so on. Remark that the result of MM_4x4_TRANSPOSE_PD is *not* invariant to the permutation of its parameters. Further, we note that MM_4x4_TRANSPOSE_PD is an involution. In general, register transpose is a relatively expensive operation due to the high aggregate latency of the vector instructions (VPERM2F128 and VSHUFPD) involved in its implementation. However, it will be shown in the following subsection that the special structure of the rank-1 update will simplify and cheapen the general register transpose through the use of optimal packing layouts in the GEMM operation.

4.2 The Quaternion Microkernel and Amortization of Register Transpose

Given that AVX only supports 16 YMM registers, the largest register block (Eq. (28)) which allows for C_r, $\vec{a}_r{}^{(\kappa)}$ and $\vec{b}_{r(\kappa)}$ to all reside in registers simultaneously is given by $N_r = M_r = 2$. As such, the quaternion microkernel must perform a sum over 2x2 rank-1 updates to update a register block of C. A single 2x2 rank-1 update requires 4 product evaluations given by

$$\begin{bmatrix} C_{r,11} \\ C_{r,12} \\ C_{r,21} \\ C_{r,22} \end{bmatrix} = \begin{bmatrix} C_{r,11} \\ C_{r,12} \\ C_{r,21} \\ C_{r,22} \end{bmatrix} + \begin{bmatrix} \vec{a}_{r,1} \\ \vec{a}_{r,1} \\ \vec{a}_{r,2} \\ \vec{a}_{r,2} \end{bmatrix} \circ \begin{bmatrix} \vec{b}_{r,1} \\ \vec{b}_{r,2} \\ \vec{b}_{r,1} \\ \vec{b}_{r,2} \end{bmatrix}, \tag{33}$$

where we have dropped the (κ) super– and subscripts for brevity. Per the discussion of the previous subsection, these product evaluations may be performed simultaneously using SIMD vector instructions given that the register data arrangement adheres to the structure Eq. (31) via Eq. (32). On top of the 32 vector instructions requires to perform the product accumulations, the general scheme for register transpose depicted in Fig. 2 requires an additional 16 register operations: 8 for transposing the components of $\vec{a}_r$ and $\vec{b}_r$, respectively. The operation overhead is further compounded by the fact that the microkernel performs many (K_c) rank-1 updates successively, thus this scheme costs $16K_c$ additional operations over the execution of the microkernel. However, such a general approach for register transpose would only be required for 4 *unique* quaternions, whereas the 4 (unique) products required for the evaluation of Eq. (33) only involve 2 sets of 2 unique quaternions. As such, simplifications to the general register transpose scheme of Fig. 2 may be made in this case.

There are two special cases for register transpose which we must consider for Eq. (33), namely those which represent the data ordering of the elements of $\vec{a}_r$ and $\vec{b}_r$, respectively:

$$(\mathsf{A}^0, \mathsf{A}^1, \mathsf{A}^2, \mathsf{A}^3) \leftarrow \texttt{MM\_4x4\_TRANSPOSE\_PD}(\mathsf{A}_1, \mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_2), \tag{34}$$

$$(\mathsf{B}^0, \mathsf{B}^1, \mathsf{B}^2, \mathsf{B}^3) \leftarrow \texttt{MM\_4x4\_TRANSPOSE\_PD}(\mathsf{B}_1, \mathsf{B}_2, \mathsf{B}_1, \mathsf{B}_2). \tag{35}$$

Here, A_1 and A_2 hold the components of $\vec{a}_{r,1}$ and $\vec{a}_{r,2}$, respectively, and similarly for B_1 and B_2 for the elements of $\vec{b}_r$. The YMM registers $\{\mathsf{A}^i\}$ and $\{\mathsf{B}^i\}$ represent the components of $\vec{a}_r$ and $\vec{b}_r$ in the order which they were passed, i.e. A^0 has the layout $[\vec{a}^0_{r,1}; \vec{a}^0_{r,1}; \vec{a}^0_{r,2}; \vec{a}^0_{r,2}]$ while B^0 has the layout $[\vec{b}^0_{r,1}; \vec{b}^0_{r,2}; \vec{b}^0_{r,1}; \vec{b}^0_{r,2}]$ and so on. The presence of redundancies in the register transpose allows for factorization of MM_4x4_TRANSPOSE_PD into the convolution of two simpler operations:

$$(\mathsf{A}^0, \mathsf{A}^1, \mathsf{A}^2, \mathsf{A}^3) \leftarrow \texttt{ATRANS2}(\texttt{ATRANS1}(\mathsf{A}_1, \mathsf{A}_2)), \tag{36}$$

$$(\mathsf{B}^0, \mathsf{B}^1, \mathsf{B}^2, \mathsf{B}^3) \leftarrow \texttt{BTRANS2}(\texttt{BTRANS1}(\mathsf{B}_1, \mathsf{B}_2)). \tag{37}$$

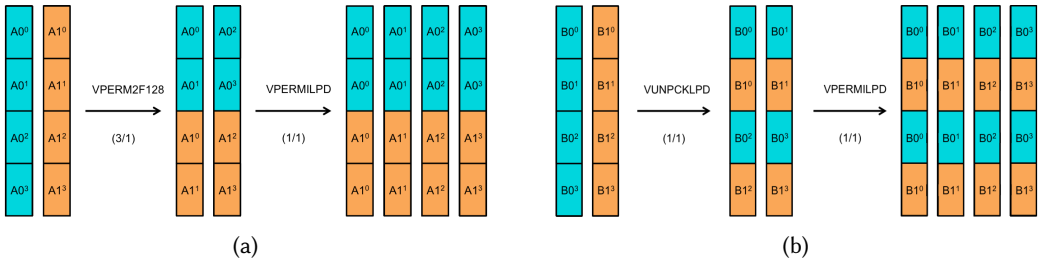


Fig. 3. Alternative register transpose schemes to efficiently handle redundancies in the general 4x4 scheme depicted in Fig. 2. Figure 3a handles the transpose case for the elements of $\vec{a}_r{}^{(\kappa)}$, and Fig. 3b for the elements of $\vec{b}_{r(\kappa)}$. Both of these schemes decompose the general register transpose operation into two operations; the first of which is space preserving. The labels beneath the arrows indicate the latency / reciprocal throughput for the instruction on the Intel(R) Sandy-Bridge microarchitechture.

An illustration of this state of affairs is given in Fig. 3 with Figs. 3a and 3b depicting the transpose of elements of $\vec{a}_r$ and $\vec{b}_r$, respectively. The first step of Fig. 3a demonstrates the effect of ATRANS1 and the second the effect of ATRANS2, and similarly for Fig. 3b. The most important aspect of the alternative transpose schemes depicted in Fig. 3 is that both ATRANS1 and BTRANS1 are space preserving. As such, they may be factored into the packing scheme as discussed in Sec. 3.3, leading to an amortization of register operations over data movement from RAM. Further, in the case of ATRANS1, not only does this procedure reduce the number of instructions which must be issued from inside the microkernel loop, it does so in a way that the most *expensive* (highest latency) register operations required for the transpose are amortized in the packing procedure. In the context of Algorithms 4 and 5, this factorization may be accounted for by setting PACKOP1 $= \alpha *$ ATRANS1 and PACKOP2 $=$ BTRANS1. Utilizing this packing strategy, the operation overhead for performing register transpose from within the microkernel is reduced by a factor of 3/4 ($16K_c$ to $4K_c$). Algorithm 6 outlines the general structure for the AVX optimized HGEMM microkernel. The following section demonstrates its performance.

5 IMPLEMENTATION AND PERFORMANCE RESULTS

HGEMM, as described in the previous section, has been implemented in the quaternion BLAS (HBLAS) component of the HAXX library. HAXX (**H**amilton's Quaternion **A**lgebra for C**XX**) [Willams-Young 2019] is a modern C++ software infrastructure developed to enable efficient scalar and linear algebra operations using quaternion and mixed–type (quaternion–complex, quaternion–real) arithmetic. As of this work, HAXX provides reference and optimized serial implementations for a representative subset of BLAS-1,2,3 functionality. For the optimized implementation of HGEMM in HAXX, the arithmetic microkernel has been implemented using C++ vector–intrinsics rather than the assembly implementations which have become ubiquitous in high–performance implementation of DGEMM and ZGEMM. This has been done primarily for the fact that vector intrinsics offer a reasonable balance between transparency in the code-base and potential performance from low-level access to assembly instructions, even if this transparency comes at a slight performance degradation. In this section, we provide performance results for the reference and optimized HGEMM implementations in HAXX for the AVX microarchitechture. All timing results were obtained using an Intel(R) Xeon(R) CPU E5-2660 (Sandy Bridge) @ 2.20 GHz (max 3.0 GHz). The E5-2660 processor admits cache sizes of 32 kB, 256 kB and 20 MB for the L1, L2, and L3 caches respectively. The L3 cache is shared among all cores on the CPU. Theoretical (serial) peak

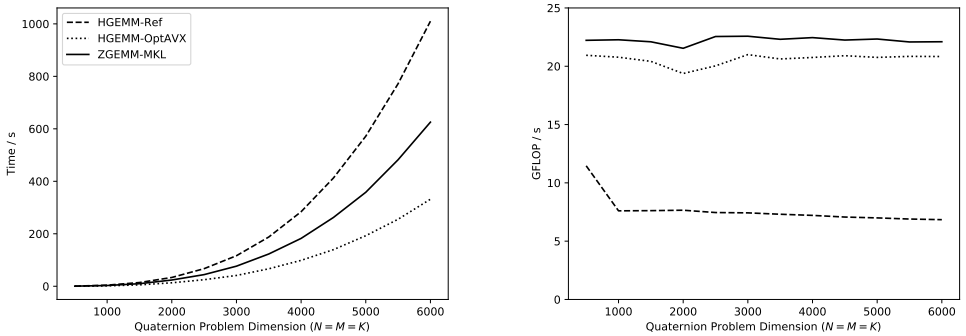
Algorithm 6: AVX Optimized HGEMM Microkernel ($N_r = M_r = 2$)

Input : Columns of packed matrices $\tilde{a}_p \in \mathbb{V}_{2K_c}(\mathbb{H})$, $\tilde{b}_p \in \mathbb{V}_{2K_c}(\mathbb{H})$,
Register block $C_r \in \mathbb{M}_2(\mathbb{H})$ of C.
Output : Updated C_r

Stream $C_{r,11}, C_{r,12}, C_{r,21}, C_{r,22}$ into registers $\mathsf{R}_{11}, \mathsf{R}_{12}, \mathsf{R}_{21}, \mathsf{R}_{22}$ from RAM
$(\mathsf{R}^0, \mathsf{R}^1, \mathsf{R}^2, \mathsf{R}^3) \leftarrow$ `MM_TRANSPOSE_4x4_PD` $(\mathsf{R}_{11}, \mathsf{R}_{12}, \mathsf{R}_{21}, \mathsf{R}_{22})$

for $k = 1 : k_c$ **do**
  Load $\mathsf{A}_1 \leftarrow \vec{a}_{p,2k}$, $\mathsf{A}_2 \leftarrow \vec{a}_{p,2k+1}$
  Load $\mathsf{B}_1 \leftarrow \vec{b}_{p,2k}$, $\mathsf{B}_2 \leftarrow \vec{b}_{p,2k+1}$
  $(\mathsf{A}^0, \mathsf{A}^1, \mathsf{A}^2, \mathsf{A}^3) \leftarrow$ `ATRANS2` $(\mathsf{A}_1, \mathsf{A}_2)$
  $(\mathsf{B}^0, \mathsf{B}^1, \mathsf{B}^2, \mathsf{B}^3) \leftarrow$ `BTRANS2` $(\mathsf{B}_1, \mathsf{B}_2)$
  $(\mathsf{R}^0, \mathsf{R}^1, \mathsf{R}^2, \mathsf{R}^3) \leftarrow$ `HMUL`$(\{\mathsf{R}^i\}, \{\mathsf{A}^i\}, \{\mathsf{B}^i\})$
end

$(\mathsf{R}_{11}, \mathsf{R}_{12}, \mathsf{R}_{21}, \mathsf{R}_{22}) \leftarrow$ `MM_TRANSPOSE_4x4_PD` $(\mathsf{R}^0, \mathsf{R}^1, \mathsf{R}^2, \mathsf{R}^3)$
Store $\mathsf{R}_{11}, \mathsf{R}_{12}, \mathsf{R}_{21}, \mathsf{R}_{22}$ in $C_{r,11}, C_{r,12}, C_{r,21}, C_{r,22}$



(a) (b)

Fig. 4. Computational timing and scaling comparisons for reference HGEMM (HGEMM-Ref), AVX optimized HGEMM (HGEMM-OptAVX) and the serial ZGEMM of Intel MKL (ZGEMM-MKL). Figure 4a shows the raw timing comparisons and Fig. 4b shows the FLOP rate comparisons between the GEMM implementations.

performance double precision arithmetic on this CPU is 24 GFLOP/s. HAXX and all benchmark executables were compiled using the Intel(R) C++ compiler with architecture specific optimizations (‘`-xHost`’) and interprocedural optimization enabled. To obtain the caching parameters, the open–source autotuning software OpenTuner [Ansel et al. 2014] was employed. On this architecture, the optimal caching parameters were found to be $M_c = N_c = 64$ and $K_c = 1024$.

Figure 4 illustrates performance comparisons for three GEMM implementations: the reference (HGEMM-Ref) and AVX-optimized (HGEMM-OptAVX) HGEMM implementations provided in the HAXX library, and the (serial) ZGEMM implementation provided by the Intel(R) Math Kernel Library (MKL) (Version 18.0 Update 1). All timings presented are representative of Eq. (27) for square

matrices with $\alpha = 1$ and $\beta = 0$. Timings for the HGEMM implementations are for the matrix product operation on $\mathbb{M}_N(\mathbb{H})$ while those for ZGEMM are for the analogous product operation on $\mathbb{M}_{2N}(\mathbb{C})$ (see Eq. (26b)). The comparison between complex and quaternion operations are presented in this manner to demonstrate the efficacy of the quaternion operation over the complex operation for *the same arithmetic operation*, i.e. the results of these operations represent the same mathematical object (up to an isomorphism). There two primary results which are to be taken from these numerical experiments:

(1) The timing comparisons depicted in Fig. 4a illustrate that quaternion arithmetic alone is not sufficient to obtain performance leverage over tuned complex matrix multiplication. The reference HGEMM implementation is significantly less performant than the ZGEMM implementation found in MKL, while the AVX optimized HGEMM implementation outperforms the ZGEMM operation by roughly a factor of 2 (as would be expected from Tab. 2). Further, as was described in Sec. 3, the reference HGEMM implementation performs significantly under the theoretical peak performance (~7 GFLOP/s vs 24 GFLOP/s) due to the algorithm being memory bound.
(2) Despite a slight difference in the FLOP rate in the GEMM implementations depicted in Fig. 4b (~22 GFLOP/s for ZGEMM-MKL and ~21 GFLOP/s for HGEMM-OptAVX), the optimized HGEMM implementation consistently outperforms the optimized ZGEMM implementation even for large matrices ($N > 3000$).

6 CONCLUSIONS

In this work, we have demonstrated the efficacy and potential of high–performance quaternion linear algebra to leverage performance increases over complex linear algebra for special class of matrices through the efficient implementation of the quaternion matrix product. The software development proposed in this work extends the existing theory of high-performance serial matrix multiplication for use with explicitly quaternion arithmetic, as outlined in Secs. 3 and 4. A series of numerical experiments given in Sec. 5 have illustrated performance comparisons between reference quaternion, optimized quaternion, and vendor tuned complex GEMM implementations. It was shown that exploitation of quaternion arithmetic alone is not sufficient to outperform high-performance implementations of complex GEMM and that analogous implementations of high-performance quaternion GEMM are necessary to leverage such improvements. Further, it was shown that even in the presence of slight difference the FLOP rate comparisons, the optimized implementation of quaternion GEMM outperforms the optimized implementation of complex GEMM for the analogous arithmetic operation. We note for completeness that while Intel(R) Sandy Bridge and the AVX instruction set are not contemporary in and of themselves, they representative of more contemporary architectures such as the Intel(R) Haswell and AMD(R) Excavator architectures which support the AVX2 instruction set. In the context of the GEMM operation, the primary feature introduced in these architectures is FMA arithmetic instructions. With the exception of architectures which support the AVX-512 instruction set (such as Intel(R) Skylake-X and Intel(R) Knight's Landing), the SIMD vector units on architectures which support either the AVX or AVX2 instruction sets are 256 bits in length. Thus with the exception of the arithmetic kernel (Eq. (31)) and the optimal values of the caching parameters, the remainder of the findings in this work would would be invariant between AVX and AVX2. In summary, the optimized implementation of quaternion GEMM provided by the HAXX library was shown to outperform its MKL optimized complex analogue by roughly a factor of 2 (as would be expected from the discussion in Sec. 2.2). As the architecture on which the numerical experiments were performed is a representative example of modern HPC architectures in general, the results presented in this work would translate to other

architectures given that one provides optimized versions of the GEMM kernels for the architecture in question.

As the matrix product is the fundamental building block for the development of important operations such as eigendecomposition and matrix factorization, its efficient implementation is a necessary condition for high-performance linear algebra software. In order for quaternion linear algebra to be a viable alternative to complex linear algebra in problems which it may be applied, optimized implementations of quaternion operations which outperform their complex counterparts must be developed. Although the power of the *theory* of quaternion algebra in the context of scientific theory and computation has been known for some time, prior to this work, no performant implementation of quaternion linear algebra has been available. It is our hope that the software developments presented in this work will aid and spark interest in the future development of high-performance quaternion linear algebra such that the full power of quaternion arithmetic may be leveraged in computationally intensive fields such as scientific computing and image processing.

ACKNOWLEDGMENTS

In the development of HAXX, DWY was supported by a fellowship from The Molecular Sciences Software Institute under National Science Foundation grant ACI-1547580. The development of HAXX has also been supported through the development of the open source Chronus Quantum supported by the National Science Foundation (OAC-1663636 to XL). This work has been further supported in part by the U.S. Department of Energy, Office of Science, Basic Energy Sciences, under Award LAB 17-1775, as part of the Computational Chemical Sciences Program.

The authors would like to thank Edward Valeev and Benjamin Pritchard for insightful discussions regarding microarchitechture optimizations and high-performance matrix multiplication and Wissam Sid Lakhdar for aid in the tuning of HAXX. Further, the authors would like to thank Benjamin Pritchard, Wissam Sid Lakhdar, Joseph Kasper and the anonymous reviewers for reviewing the content of the manuscript and providing meaningful insight.

REFERENCES

Stephen L. Adler. 1995. *Quaternionic Quantum Mechanics and Quantum Fields.* Oxford University Press, Oxford, U.K.

Ramesh C. Agarwal, Fred G. Gustavson, and Mohammad Zubair. 1994. Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms. *IBM Journal of Research and Development* 38, 5 (1994), 563–576. https://doi.org/10.1147/rd.385.0563

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898719604

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14).* ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

Markus K Armbruster. 2017. Quaternionic Formulation of the Two-Component Kohn-Sham Equations and Efficient Exploitation of Point Group Symmetry. *J. Chem. Phys.* 147, 5 (2017), 054101. https://doi.org/10.1063/1.4995614

Vladimir Igorevich Arnol'd. 1995. The Geometry of Spherical Curves and The Algebra of Quaternions. *Russian Mathematical Surveys* 50, 1 (1995), 1–68. https://doi.org/10.1070/RM1995v050n01ABEH001662

Andrew Baker. 1999. Right Eigenvalues for Quaternionic Matrices: A Topological Approach. *Linear Algebra and Its Applications* 286, 1-3 (1999), 303–309. https://doi.org/10.1016/S0024-3795(98)10181-7

L. Susan Blackford, James Demmel, Jack J. Dongarra, Iain Duff, Sven Hammarlind, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Rolan Pozo, Karin Remington, and R. Clint Whaley. 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Soft.* 28, 2 (2002), 135–151. https://doi.org/10.1145/567806.567807

Angelika Bunse-Gerstner, Ralph Byers, and Volker Mehrmann. 1989. A Quaternion QR Algorithm. *Numer. Math.* 55, 1 (1989), 83–95. https://doi.org/10.1007/BF01395873

Arthur Cayley. 1845. XIII. On Certain Results Relating to Quaternions: To the Editors of the Philosophical Magazine and Journal. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 26, 171 (1845), 141–145. https://doi.org/10.1080/14786444508562684

Claude Chevalley. 1996. *The Algebraic Theory of Spinors and Clifford Algebras: Collected Works.* Vol. 2. Springer Science & Business Media.

Cipher A. Deavours. 1973. The Quaternion Calculus. *The American Mathematical Monthly* 80, 9 (1973), 995–1008. https://doi.org/10.2307/2318774

Jack J. Dongarra, Jermey Du Cruz, Sven Hammarling, and Iain S. Duff. 1990. Algorithm 679: A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Soft.* 16, 1 (1990), 18–28. https://doi.org/10.1145/77626.77627

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. Algorithm 656: An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Soft.* 14, 1 (1988), 18–32. https://doi.org/10.1145/42288.42292

Jack J. Dongarra, John R. Gabriel, Dale D. Koelling, and James H. Wilkinson. 1984. Solving the Secular Equation Including Spin Orbit Coupling for Systems with Inversion and Time Reversal Symmetry. *J. Comput. Phys.* 54, 2 (1984), 278–288. https://doi.org/10.1016/0021-9991(84)90119-0

Ulf Ekström, Patrick Norman, and Vincenzo Carravetta. 2006. Relativistic Four-Component Static-Exchange Approximation for Core-Excitation Processes in Molecules. *Phys. Rev. A* 73, 2 (2006), 022501. https://doi.org/10.1103/PhysRevA.73.022501

Todd A Ell, Nicolas Le Bihan, and Stephen J Sangwine. 2014. *Quaternion Fourier Transforms for Signal and Image Processing.* John Wiley & Sons.

Agner Fog. 2012. The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. *Copenhagen University College of Engineering* (2012), 2–29.

Georg Frobenius. 1878. *Über lineare Substitutionen und bilineare Formen.* Reimer.

Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Soft.* 34, 3 (2008), 12. https://doi.org/10.1145/1356052.1356053

Carl Grubin. 1970. Derivation of the Quaternion Scheme via the Euler Axis and Angle. *Journal of Spacecraft and Rockets* 7, 10 (1970), 1261–1263. https://doi.org/10.2514/3.30149

John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. 2001. A Family of High-Performance Matrix Multiplication Algorithms. In *International Conference on Computational Science.* Springer, 51–60.

Brian Hall. 2015. *Lie Groups, Lie Algebras, and Representations: An Elementary Introduction.* Vol. 222. Springer.

William Rowan Hamilton. 1866. *Elements of Quaternions.* Longmans, Green, & Company.

Johan Henriksson, Patrick Norman, and Hans Jørgen Aa Jensen. 2005. Two-Photon Absorption in the Relativistic Four-Component Hartree–Fock approximation. *J. Chem. Phys.* 122, 11 (2005), 114106. https://doi.org/10.1063/1.1869469

Heinz Hopf. 1931. Über die Abbildungen der dreidimensionalen Sphäre auf die Kugelfläche. *Math. Ann.* 104, 1 (1931), 637–665. https://doi.org/10.1007/BF01457962

Lawrence P. Horwitz and Lawrence C. Biedenharn. 1984. Quaternion quantum mechanics: Second quantization and gauge fields. *Annals of Physics* 157, 2 (1984), 432 – 488. https://doi.org/10.1016/0003-4916(84)90068-X

Adolf Hurwitz. 1922. Über die Komposition der quadratischen Formen. *Math. Ann.* 88, 1-2 (1922), 1–25. https://doi.org/10.1007/bf01448439

Drahoslava Janovská and Gerhard Opfer. 2003. Givens' Transformation Applied to Quaternion Valued Vectors. *BIT Numerical Mathematics* 43, 5 (2003), 991–1002. https://doi.org/10.1023/B:BITN.0000014561.58141.2c

Zhigang Jia, Musheng Wei, Mei-Xiang Zhao, and Yong Chen. 2018. A New Real Structure-Preserving Quaternion QR Algorithm. *J. Comput. Appl. Math.* 343 (2018), 26–48. https://doi.org/10.1016/j.cam.2018.04.019

Lukas Konecny, Marius Kadek, Stanislav Komorovsky, Kenneth Ruud, and Michal Repisky. 2018. Resolution-of-Identity Accelerated Relativistic Two-and Four-Component Electron Dynamics Approach to Chiroptical Spectroscopies. *J. Chem. Phys.* 149, 20 (2018), 204104. https://doi.org/10.1063/1.5051032

Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic Linear Algebra Subprograms for FORTRAN Usage. *ACM Trans. Math. Soft.* 5, 3 (1979), 308–323.

Nicolas Le Bihan and Jérôme Mars. 2004. Singular Value Decomposition of Quaternion Matrices: A New Tool for Vector-Sensor Signal Processing. *Signal Processing* 84, 7 (2004), 1177–1199.

Nicolas Le Bihan and Stephen J Sangwine. 2003. Quaternion Principal Component Analysis of Color Images. In *ICIP 2003. Proceedings. 2003 International Conference on Image Processing, 2003.*, Vol. 1. IEEE, I–809.

Ying Li, Musheng Wei, Fengxia Zhang, and Jianli Zhao. 2019. On The Power Method for Quaternion Right Eigenvalue Problem. *J. Comput. Appl. Math.* 345 (2019), 59–69. https://doi.org/10.1016/j.cam.2018.06.015

Terry A. Loring. 2012. Factorization of Matrices of Quaternions. *Expositiones Mathematicae* 30, 3 (2012), 250–267. https://doi.org/10.1016/j.exmath.2012.08.006

Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Soft.* 43, 2 (Aug. 2016), 12:1–12:18. https://doi.org/10.1145/2925987

Santiago Arribas Mocoroa, Antonio Elipe, and Manuel Palacios Latasa. 2006. Quaternions and the rotation of a rigid body. *Celestial Mechanics and Dynamical Astronomy* 96, 3-4 (2006), 239–251. https://doi.org/10.1007/s10569-006-9037-6

Masahiko Nakano, Junji Seino, and Hiromi Nakai. 2017. Development of Spin-Dependent Relativistic Open-Shell Hartree–Fock Theory with Time-Reversal Symmetry (I): The Unrestricted Approach. *Int. J. Quant. Chem.* 117, 10 (2017), e25356. https://doi.org/10.1002/qua.25356

Daoling Peng, Jianyi Ma, and Wenjian Liu. 2009. On the Construction of Kramers Paired Double Group Symmetry Functions. *Int. J. Quant. Chem.* 109, 10 (2009), 2149–2167. https://doi.org/10.1002/qua.22078

Leiba Rodman. 2014. *Topics in Quaternion Linear Algebra.* Princeton University Press.

Trond Saue, Knut Fægri Jr., Trygve Helgaker, and Odd Gropen. 1997. Principles of Direct 4-Component Relativistic SCF: Application to Caesium Auride. *Mol. Phys.* 91, 5 (1997), 937–950. https://doi.org/10.1080/002689797171058

Trond Saue and Trygve Helgaker. 2002. Four-Component Relativistic Kohn–Sham Theory. *J. Comput. Chem.* 23, 8 (2002), 814–823. https://doi.org/10.1002/jcc.10066

Toru Shiozaki. 2017. An Efficient Solver for Large Structured Eigenvalue Problems in Relativistic Quantum Chemistry. *Mol. Phys.* 115, 1-2 (2017), 5–12. https://doi.org/10.1080/00268976.2016.1158423

Ken Shoemake. 1985. Animating Rotation with Quaternion Curves. *ACM SIGGRAPH computer graphics* 19, 3 (1985), 245–254.

Jason L. Stuber and Joesf Paldus. 2003. Symmetry Breaking in the Independent Particle Model. In *Fundamental World of Quantum Chemistry: A Tribute to the Memory of Per-Olov Löwdin.* Kluwer Academic Publishers, 67–139.

Anthony Sudbery. 1979. Quaternionic analysis. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 85. Cambridge University Press, 199–225. https://doi.org/10.1017/S0305004100055638

Field G. Van Zee and Tyler Smith. 2017. Implementing High-performance Complex Matrix Multiplication via the 3m and 4m Methods. *ACM Trans. Math. Soft.* 44, 1 (July 2017), 7:1–7:36. https://doi.org/10.1145/3086466

Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn. 2016. The BLIS Framework: Experiments in Portability. *ACM Trans. Math. Soft.* 42, 2 (June 2016), 12:1–12:19. https://doi.org/10.1145/2755561

Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Soft.* 41, 3 (June 2015), 14:1–14:33. https://doi.org/10.1145/2764454

Lucas Visscher and Trond Saue. 2000. Approximate Relativistic Electronic Structure Methods Based on the Quaternion Modified Dirac Equation. *J. Chem. Phys.* 113 (2000), 3996. https://doi.org/10.1063/1.1288371

Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* IEEE, 1–12.

R. Clinton Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *IEEE/ACM Conference on Supercomputing, 1998.* IEEE, 38–38.

R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated Empirical Optimizations of Software and the ATLAS project. *Parallel Comput.* 27, 1-2 (2001), 3–35. https://doi.org/10.1016/S0167-8191(00)00087-9

David B. Willams-Young. 2019. HAXX: Hamilton's Quaternion Algebra for CXX. https://github.com/wavefunction91/HAXX

Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-Driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems.* IEEE, 684–691.

Rui Zeng, Jiasong Wu, Zhuhong Shao, Yang Chen, Beijing Chen, Lotfi Senhadji, and Huazhong Shu. 2016. Color Image Classification via Quaternion Principal Component Analysis Network. *Neurocomputing* 216 (2016), 416–428. https://doi.org/10.1016/j.neucom.2016.08.006

Fuzhen Zhang. 1997. Quaternions and Matrices of Quaternions. *Linear Algebra and Its Applications* 251 (1997), 21–57. https://doi.org/10.1016/0024-3795(95)00543-9